Assignment 3

Exercise 3.3 Write out the **rightmost derivation** of the string below from the expression grammar at the end of Sect. 3.6.5, corresponding to ExprPar.fsy. Take note of the sequence of grammar rules (A–I) used.

```
Main ::= Expr EOF                          rule A
Expr ::= NAME                              rule B
       |  CSTINT                           rule C
       |  MINUS CSTINT                     rule D
       |  LPAR Expr RPAR                   rule E
       |  LET NAME EQ Expr IN Expr END     rule F
       |  Expr TIMES Expr                  rule G
       |  Expr PLUS  Expr                  rule H
       |  Expr MINUS Expr                  rule I
```

(the expression grammar at the end of Sect. 3.6.5)

let z = (17) in z + 2 * 3 end EOF

Rule A) **Expr** EOF ⇒

Rule F) LET NAME EQ Expr IN Expr **END** EOF ⇒

sub) LET NAME EQ Expr IN **Expr** end EOF ⇒

Rule G) LET NAME EQ Expr IN Expr TIMES **Expr** end EOF ⇒

Rule C) LET NAME EQ Expr IN Expr TIMES **CSTINT** end EOF ⇒

sub) LET NAME EQ Expr IN Expr **TIMES** 3 end EOF ⇒

sub) LET NAME EQ Expr IN **Expr** * 3 end EOF ⇒

Rule H) LET NAME EQ Expr IN Expr PLUS **Expr** * 3 end EOF ⇒

Rule C) LET NAME EQ Expr IN Expr PLUS **CSTINT** * 3 end EOF ⇒

sub) LET NAME EQ Expr IN Expr **PLUS** 2 * 3 end EOF ⇒

sub) LET NAME EQ Expr IN **Expr** + 2 * 3 end EOF ⇒

Rule B) LET NAME EQ Expr IN **NAME** + 2 * 3 end EOF ⇒

sub) LET NAME EQ Expr **IN** z + 2 * 3 end EOF ⇒

sub) LET NAME EQ **Expr** in z + 2 * 3 end EOF ⇒

Rule E) LET NAME EQ LPAR Expr **RPAR** in z + 2 * 3 end EOF ⇒

sub) LET NAME EQ LPAR **Expr**) in z + 2 * 3 end EOF ⇒

Rule C) LET NAME EQ LPAR **CSTINT**) in z + 2 * 3 end EOF ⇒

sub) LET NAME EQ **LPAR** 17) in z + 2 * 3 end EOF ⇒

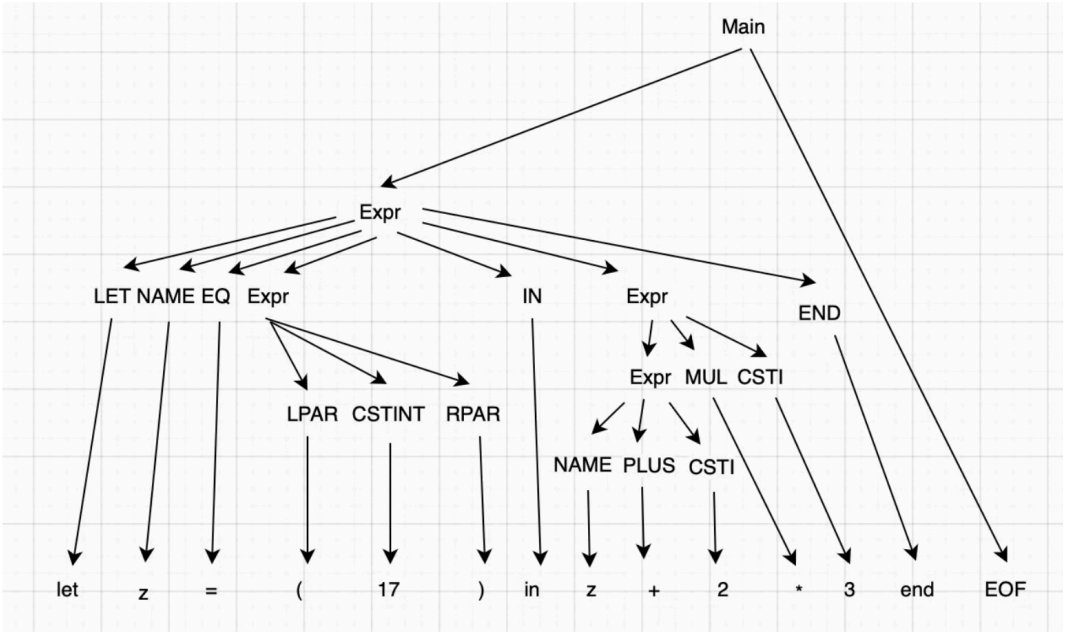
sub) LET NAME **EQ** (17) in z + 2 * 3 end EOF ⇒

sub) LET **NAME** = (17) in z + 2 * 3 end EOF ⇒

sub) **LET** z = (17) in z + 2 * 3 end EOF

sub) let z = (17) in z + 2 * 3 end EOF

Exercise 3.4 Draw the above derivation as a tree.



Exercise 3.5 Get expr.zip from the book homepage and unpack it. Using a command prompt, generate (1) the lexer and (2) the parser for expressions by running fslex and fsyacc; then (3) load the expression abstract syntax, the lexer and parser modules, and the expression interpreter and compilers, into an interactive F# session (fsharpi):

```
fsyacc --module ExprPar ExprPar.fsy
fslex --unicode ExprLex.fsl
dotnet fsi -r ~/fsharp/FsLexYacc.Runtime.dll Absyn.fs ExprPar.fs ExprLex.fs Parse.fs Expr.fs
```

Now try the parser on several example expressions, both well-formed and ill-formed ones, such as these, and some of your own invention:

```
open Parse;;
fromString "1-2- 3";;
fromString "1 + -2";;
fromString "x++";; (* failed *)
fromString "1 + 1.2";; (* failed *)
fromString "1 + ";; (* failed *)
fromString "let z = (17) in z +2*3 end";;
fromString "let z = (17) inz+2*3 end";; (* failed *)
fromString "let z = 17) in z+2*3 end";; (* failed *)
fromString "let in = (17) in z+2*3 end";; (* failed *)
fromString "1 + let x=5 in let y=7+x in y+y end + x end";;
```

Exercise 3.6 Use the expression parser from Parse.fs and the compiler `scomp` (from expressions to stack machine instructions) and the associated datatypes from `Expr.fs`, to define a function `compString : string -> sinstr list`that parses a string as an expression and compiles it to stack machine code.

Written in bottom of `Expr.fs`

Exercise 3.7 Extend the expression language abstract syntax and the lexer and parser specifications with conditional expressions. The abstract syntax should be `If(e1, e2, e3)`, so modify file `Absyn.fs` as well as `ExprLex.fsl` and file `ExprPar.fsy`. The concrete syntax may be the keyword-laden F#/ML-style:

```
if e1 then e2 else e3
```

Written in `Absyn.fs, ExprLex.fsl, ExprPar.fsy`